IDA PAPER P-2765

ANALYSIS AND GUIDELINES FOR REUSABLE ADA SOFTWARE

David A. Wheeler
Dennis W. Fife, *Task Leader*

August 1992

Prepared for
Strategic Defense Initiative Organization (SDIO)

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Group Reports

Group Reports record the findings and results of IDA established working groups and panels composed of senior individuals addressing major issues which otherwise would be the subject of an IDA Report. IDA Group Reports are reviewed by the senior individuals responsible for the project and others as selected by IDA to ensure their high quality and relevance to the problems studied, and are released by the President of IDA.

Papers

Papers, also authoritative and carefully considered products of IDA, address studies that are narrower in scope than those covered in Reports. IDA Papers are reviewed to ensure that they meet the high standards expected of refereed papers in professional journals or formal Agency reports.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts (a) to record substantive work done in quick reaction studies, (b) to record the proceedings of conferences and meetings, (c) to make available preliminary and tentative results of analyses, (d) to record data developed in the course of an investigation, or (e) to forward information that is essentially unanalyzed and unevaluated. The review of IDA Documents is suited to their content and intended use.

The work reported in this document was conducted under contract MDA 903 89 C 0003 for the Department of Defense. The publication of this IDA document does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1992	Final

4. TITLE AND SUBTITLE
Analysis and Guidelines for Reusable Ada Software

5. FUNDING NUMBERS
MDA 903 89 C 0003

T-R2-597.2

6. AUTHOR(S)
David A. Wheeler

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for Defense Analyses (IDA)
1801 N. Beauregard St.
Alexandria, VA 22311-1772

8. PERFORMING ORGANIZATION REPORT NUMBER
IDA Paper P-2765

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
SDIO/GMI
The Pentagon, Room 1E149
Washington, DC 20301-7100

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, unlimited distribution: 19 October 1992

12b. DISTRIBUTION CODE
2A

13. ABSTRACT (Maximum 200 words)

SDIO is developing the Global Protection Against Limited Strikes (GPALS) system. Much of the software is to be developed in Ada, and reuse will be considered in the software design and implementation phases. This document provides an analysis and candidate guidelines for developing reusable Ada software to supplement existing SDIO guidelines. The guidelines in this document are based on previous IDA work, IDA Paper P-2378, *An Approach for Constructing Reusable Components in Ada* by Stephen Edwards. In addition, two other sources were examined: *GPALS Software Standards* by GE Aerospace, and *Ada Quality and Style: Guidelines for Professional Programmers* by the Software Productivity Consortium. The analysis shows that many of Edwards's guidelines are already included in *Ada Quality and Style* and thus will already be used by SDIO. Some of the guidelines require further research before inclusion in a candidate set of guidelines. The remaining guidelines are presented as candidate guidelines in a form similar to that of *Ada Quality and Style*. This document presents the candidate guidelines in a form which could be readily added to existing Ada development guidelines.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Ada; Software Reuse; Guidelines; 3C Model; Global Protection Against Limited Strikes (GPALS)	61
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

IDA PAPER P-2765

ANALYSIS AND GUIDELINES FOR REUSABLE ADA SOFTWARE

David A. Wheeler
Dennis W. Fife, *Task Leader*

August 1992

INSTITUTE FOR DEFENSE ANALYSES
Contract MDA 903 89 C 0003
Task T-R2-597.2

PREFACE

The Institute for Defense Analyses (IDA) was tasked by the Strategic Defense Initiative Organization (SDIO) to "provide a candidate Global Protection Against Limited Strikes (GPALS) standard for maintainable and reusable software based on the 3C model." This work was done under Contract MDA 903 89 C 0003, Task Order 597.2, Amendment 4, SDIO Software Technology Plan. This document provides an analysis and a set of candidate guidelines for developing reusable Ada software to supplement existing guidelines already required by SDIO.

The guidelines are based on previous IDA work and extend the GE Aerospace's *GPALS Software Standards* and *Ada Quality and Style: Guidelines for Professional Programmers*, written by the Software Productivity Consortium. The guidelines in this document are written in a form to be readily added to existing Ada development guidelines.

The document is directed towards the developers of reusable Ada software, in particular those who are developing GPALS software.

This document was reviewed by the following members of IDA: Dr. Richard J. Ivanetich, Mr. Robert J. Knapper, Dr. Reginald N. Meeson, Mr. Clyde G. Roby, and Mr. Jonathan D. Wood. Their contributions are gratefully acknowledged.

EXECUTIVE SUMMARY

The Strategic Defense Initiative Organization (SDIO) is developing the Global Protection Against Limited Strikes (GPALS) system. Much of this software is to be developed in Ada, and the software design and implementation phases will be considering reuse. SDIO has already chosen some guidelines for developing reusable components to assist in this effort, but no one had considered merging other guidelines from a separate SDIO-sponsored effort into this selected set of guidelines.

This document provides an analysis and candidate guidelines for developing reusable Ada software to supplement those already required by SDIO. These guidelines are based on previous IDA work by Edwards [1990], which in turn is based on the 3C model as documented by Tracz [1989]. The 3C model was developed at the Reuse In Practice Workshop in 1989. The name of the 3C model comes from the names of the three ideas on which it is based: concept, content, and context. To create a reusable component using the 3C model, a designer separates *what* the component will do (the concept), *how* it will do it (the content), and *what external information* is necessary to tailor the component for use (the context).

Two other documents form the basis for this document. The first is the Software Productivity Consortium (SPC)'s *Ada Quality and Style: Guidelines for Professional Programmers* which includes a number of guidelines for developing reusable Ada components. The second is General Electric Aerospace's *GPALS Software Standards*, which defines the GPALS software development guidelines and includes by reference SPC's document.

The analysis shows that many of Edwards' guidelines are already included in *Ada Quality and Style* and thus will already be used by SDIO. Some of the guidelines require further research before inclusion in a candidate set of guidelines. The remaining guidelines are presented as candidate guidelines in a form similar to that of *Ada Quality and Style*.

TABLE OF CONTENTS

LIST OF TABLES

1. INTRODUCTION

1.1 PURPOSE

This document provides an analysis and set of candidate guidelines for developing reusable Ada software. This analysis and set of candidate guidelines are based on previous IDA work by Edwards [1990], which in turn is based on the 3C model as developed at the Reuse In Practice Workshop in 1989 and documented by Tracz [1989]. These candidate guidelines are intended to supplement either *Ada Quality and Style: Guidelines for Professional Programmers*[1] by the Software Productivity Consortium (SPC) [1991] or the *GPALS Software Standards* by General Electric Aerospace (GE). These candidate guidelines are intended for developers of reusable Ada software for the Global Protection Against Limited Strikes (GPALS) system, as well as developers of reusable Ada software for other projects.

1.2 SCOPE

The analyzed guidelines are extracted from Institute for Defense Analyses (IDA) Paper P-2378, *An Approach for Constructing Reusable Components in Ada* [Edwards 1990]. Guidelines which were found to overlap *Ada Quality and Style*, overlap each other, or appeared to require further study, have been separated from the rest of the guidelines. The remaining guidelines are presented as a candidate set of guidelines and are presented in a form that is easier than the previous IDA paper for software developers to use.

These guidelines concentrate on reusable code as a product, not on the process of developing reusable code.

The developer must determine which guidelines may be inappropriate for a specific application. The rationale for each guideline has been included in this document to help developers determine the best trade-off between the aim of each guideline and the user's needs. The text of the candidate guidelines include their rationale and the qualitative impact of following or not following them, but no cost/benefit study has been performed for this candidate set of guidelines.

1.3 BACKGROUND

Three documents serve as the basis of this document:

---

1. For the rest of this paper this title is shortened to *Ada Quality and Style*.

- *Ada Quality and Style: Guidelines for Professional Programmers* by the Software Productivity Consortium (SPC) [1991].
- *GPALS Software Standards* by General Electric Aerospace (GE) [1991].
- *An Approach for Constructing Reusable Components in Ada* by Stephen Edwards [1990].

Ada Quality and Style defines a set of Ada coding guidelines, including a number of guidelines for developing reusable components. The Ada Joint Program Office (AJPO) suggests this Ada style guide for use in Department of Defense programs (as announced in [AdaIC 1991a] and [AdaIC 1991b]).

The *GPALS Software Standards* document defines the required guidelines for developing software for the GPALS system [SDIO 1992]. It includes by reference *Ada Quality and Style*. As of this writing the *GPALS Software Standards* references a previous version of the *Ada Quality and Style* (1989) instead of the more recent version (1991). The more recent version of *Ada Quality and Style* (1991) will be used in this document. It is expected that an upcoming version of the *GPALS Software Standards* will reference the newer version of *Ada Quality and Style*.[2]

Edwards [1990] recommends a set of guidelines for developing reusable Ada components based on a model termed the 3C model. These guidelines are for detailed design and code, and concentrate on reusable Ada code as a product instead of the process for developing this code. Edwards identifies 21 specific guidelines, implies 2 others, and 2 guidelines can be split into 2 more guidelines each, making a total of 25 guidelines.

The 3C model was developed at the Reuse In Practice Workshop in 1989 and was first described in a working group's report by Tracz [1989]. This workshop was sponsored by IDA, SEI, SDIO, and the ACM and is described further by Baldo [1990]. The name of the 3C model comes from the names of the three ideas on which it is based: concept, content, and context. To create a reusable component using the 3C model, separate *what* the component will do (the concept), *how* it will do it (the content), and *what external information* is necessary to tailor the component for use (the context). Detailed discussion of the 3C model is presented in section 5.1 of this document.

Many of Edwards' guidelines are included in *Ada Quality and Style* and thus will already be used by SDIO. Some of Edwards' guidelines require significant further study before insertion into SDIO's development environment. Edwards includes a complex example that is good for showing how the guidelines work together but makes initial understanding of the guidelines difficult. Finally, the format of Edwards' document is not similar to that of other coding guidelines,

2. Personal communication with Axel Ahlberg of General Electric (GE), 1992.

making it much more difficult to use in conjunction with them.

For a tutorial on related software reuse issues see Tracz [1988].

1.4 DOCUMENT ORGANIZATION

Section 2 presents the approach taken to develop the candidate list of guidelines. Section 3 presents the guidelines which are already in the *Ada Quality and Style* document and thus will be used by SDIO, as well as guidelines which are overlapped by other guidelines. Section 4 presents the guidelines which require further study and the rationale for placing them in this category. Section 5 of this document presents the candidate guidelines based on the 3C model. The document closes with an acronym list and the appendices. Appendix A shows that every guideline from Edwards was considered and the category to which each guideline was allocated. Appendix B includes source code examples.

Readers who are solely interested in the candidate guidelines should move immediately to section 5 where they are presented.

2. APPROACH

In the process of analyzing Edwards' document to derive candidate guidelines, Edwards' document was first examined to find implied guidelines. This step found two additional guidelines, "model use" and "procedure variable encapsulation". The use of models was the basis of Edwards' paper but was never numbered as a specific guideline. Edwards' document also described mechanisms for procedure variable encapsulation but did not identify it as a separate guideline, and this appeared to be a promising guideline candidate.

This set of guidelines was then compared to *Ada Quality and Style* and to each other. It was found that a number of guidelines were already in *Ada Quality and Style* or subsumed by other guidelines, and these were removed from the candidate set.

This resulting list was then examined to see if any guidelines had outstanding issues which required further study and would be inappropriate for a candidate list. This examination included searching the literature, discussion with various experts, and a few prototypes to test the validity of some individual guidelines.

The five remaining guidelines are listed as candidate reuse guidelines. Since these guidelines were not in a form similar to *Ada Quality and Style*, these guidelines were then reworded and reorganized to conform to that format. Each guideline includes text which describes it and additional caveats where needed.

A list showing the set of Edwards' guidelines and how they were allocated is shown in appendix A.

3. GUIDELINES OVERLAPPING EXISTING GUIDELINES

This section presents the guidelines which are not included in the candidate set of guidelines because they overlap guidelines in *Ada Quality and Style* or other guidelines in Edwards [1990]. This section is included to show that the other guidelines are not being ignored, but instead are either already being used by SDIO or would be used if these candidate guidelines were used. Note that *Ada Quality and Style* has a section specifically identifying reuse-specific guidelines, and all of the overlapping guidelines are identified in this section.

Table 1 shows the guidelines from Edwards [1990] which overlap *Ada Quality and Style*. The three columns show the guideline number, pages, and text from Edwards [1990]. The final column shows the guideline number of an equivalent guideline in *Ada Quality and Style*.

In addition, Edwards' "minimum profile" guideline (defined as guideline 10 in Edwards) subsumes Edwards' guidelines 6, 15 and 17, so the latter three are also omitted.

There are differences between *Ada Quality and Style* and Edwards' document. Edwards' guideline 5 recommends using **limited private**, while *Ada Quality and Style* [p. 138] (private and limited private types) recommends exporting limited private, private, or nonprivate as appropriate. *Ada Quality and Style* [p. 135] requires both active and passive iterators but does not mention random-access iterators. In contrast, Edwards [1990, p. 82] has a more complex requirement: iterators are not actually required, but when included, active sequential access iterators are to be preferred over passive iterators; and when active random-access iterators are included they should be provided in addition to sequential iterators.

TABLE 1. Guidelines Overlapping Existing Guidelines

Guide-line No.	Pages	Guideline Text	SPC Location
1	29-31	A concept should be represented as a single, generic package specification.	8.3.2a, 8.3.4
2	29-31	Each concept should provide one and only one abstraction—i.e., define a single object type.	8.3.4
3	31-33	There should be *no* fixed, horizontal coupling between a concept and other concepts. In other words, Ada packages that represent reusable component concepts should not **with** other packages.	8.4.1a-c, 8.3.2b
4	35-42	Each abstraction should provide a *complete* set of basic operations. The operations provided should be sufficient for the reuser to construct any complex manipulations that are needed from them.	8.3.1
5	35-42	For the abstract types defined in a component, use **limited private**.	Related to 8.3.6b
8	35-42	All abstract types in the context (i.e. which are generic parameters in the package specification) should be **limited private**. Similarly, Initialize and Finalize operations for such a type should also be part of the generic formal parameter list.	8.3.6a
11	42-45	Each implementation of a concept should exist as a separate Ada generic package. However, all the package specifications for these implementations should be identical except for the package name and implementation context.	8.4.3

Guidelines Overlapping Existing Guidelines (Continued)

Guide-line No.	Pages	Guideline Text	SPC Location
19	70-84	If there are explicit iterators, prefer active sequential over passive; protect from structural modification; use explicit iterator state objects.	8.3.5b-d
20	70-84	If there is an active random-access iterator, also provide active sequential access. When a random access iterator is included, it should use explicit state objects, support destructive iterations, and ensure correct behavior under both nested and concurrent operations.	8.3.5b requires both

4. GUIDELINES REQUIRING FURTHER STUDY

Table 2 shows the guidelines from Edwards [1990] that require significant further study before they are included in a candidate set of guidelines.

TABLE 2. Guidelines Requiring Further Study

Guide-line Number	Pages	Guideline Text
13	45-53	Aliasing behavior (structural sharing) is the responsibility of the abstraction, not the user.
14	45-53	Use Swap, not Copy, as the basic data movement operator.
16	45-53	Design constructors and selectors using Swap, not Copy.
21	84-92	Use a standard interface for save and restore operations following the examples from Edwards [1990, pp. 90-91].
7,10	35-42	[Provide and use Initialize routines].
12	42-25	Common source should be located in a single location. The Ada package specifications and bodies for multiple implementations of a single concept should come from a common source. Possible mechanisms for doing so include the use of a preprocessor.
N/A	53-61	If procedure (or function) variables are needed, use a concept which encapsulates their implementation.

4.1 SWAPPING

Edwards' guidelines 13, 14, and 16 involve the use of swapping instead of copying as the basic data movement operation (this is termed "swapping semantics"). Strictly speaking, guideline 13 does not require swapping semantics, but implementing guideline 13 without swapping adds a layer of complexity on reusable component implementations, as shown in Edwards [1990, pp. 46-49].

Using swapping wherever possible instead of copying has advantages for reusable components, as described by Harms [1991] and Edwards [1990, pp. 49-51]. For example, with swapping it is easy to design unconstrained generic components in which no restrictions are placed on the type of items contained inside it. In addition, algorithms are potentially more efficient when copying is avoided. When implemented as pointer swaps, swapping has the advantage of constant performance no matter how complex the underlying data structure. Finally, programs may be more reliable, since bugs resulting from visible aliasing and dangling references would not occur. These are all argued further by Harms [1991, p. 434].

There are concerns, however, which need to be addressed before including these guidelines in a candidate set. Changing the basic data movement operator is a major change from existing practice and there is no experience in its use in large systems. Currently only smaller, basic objects (similar to those taught in introductory computer science courses) have been created using this approach. For an example of these components, see Weide [1986]. This appears to be a major risk in its application to large systems and suggests that its use in smaller systems should be tried first before attempting to migrate to large systems. There is some evidence (Weide [1986, p. 1]) that even well-known components are tricky to design using this approach. This approach could make designing complex components, which are difficult to design using conventional approaches, too difficult to design in a reasonable amount of time.

There are a number of other concerns about swapping:

- Swapping creates difficulties in exception handling. Exceptions are more difficult to handle since a reusable component should always attempt to bring the component back to some original state before passing the exception on. This is noted in *Ada Quality and Style* by SPC [1991, p. 129].
- Swapping introduces an additional source of error, forgetting to swap an extracted value back into its source, which is unnecessary using copy semantics. Using copy semantics, iterators do not automatically modify the structure they are iterating over and selectors provide read-only access to structures. Using swapping semantics, iterators would use accessor operations that would modify their internal structures. Selectors would not exist, and would be replaced by accessor operations that would swap the contents of the component. Code that originally simply examined a small part of a component must have additional code to swap the data back. Forgetting to swap the data back would be a new potential source of error since this would not be needed using copy semantics.
- Because swapping is a nontraditional method of data movement, there are few examples, training materials, or experts who can provide the guidance necessary to transition to this new approach. There is little data on its

limitations or the training necessary to use it effectively.

- There is little evidence that swapping scales up. There are arguments (as presented by Harms [1991] and Edwards [1990]) as to why swapping scales up to larger components better than copying. There is a small library of a few small components developed using swapping (as described in Weide [1986]). Currently we are unaware of examples of swapping-based components in anything larger than a few thousand lines of code.
- Swapping is not supported directly by Ada or other commonly used languages. This covers a number of concerns:
  - — Parameter passing in Ada is not by swapping. Ada's parameter modes (which are **in**, **out**, and **in out**) do not correspond well to the modes of parameter passing through swapping.
  - — Ada cannot detect when the same item is passed more than once in a parameter list, which is acceptable when using copy semantics but is an error when using swapping.
  - — Existing components do not always provide swap operations.
  - — Constants are not easily handled and require additional instructions in Ada. For example, pushing an integer constant on a stack would require a temporary variable in Ada when using swapping for data movement:

    ```
    temp := 5; stack.push(temp);
    ```

    In a swap-based language, these temporary variables could be automatically generated by the compiler, but Ada does not provide this automatic generation.
- Swapping may require a much more complicated implementation than the same component developed using copy-based semantics according to Edwards [1990, p. 52].

Research and experimentation on swapping should continue because it has the potential to make combining reusable components much easier than it is today. Researchers in this area include Weide [1986] and Harms [1991].

4.2 SAVE AND RESTORE

Edwards' guideline 21 specifies an interface for save and restore behavior. The idea of having a general, standard interface for save and restore behavior is a good one. However, there are some concerns regarding this interface. First, this interface is based on swapping, so the concerns above apply. Also, the interface described achieves generality with great complexity in both interface and implementation.

4.3 INITIALIZE OPERATION

The final guideline in table 2 is actually a portion of two guidelines (7 and 10) describing initialization. Edwards recommends that every concept provide an initialization procedure since some components may need such an operation. The advantage of Edwards' approach is generality and consistency—all components would have an initialization operation which others would use.

However, there are many disadvantages to requiring an initialization operation on all components:

- A separate initialization operation is often unnecessary. Ada provides a default initialization mechanism that is sufficient for many types of reusable components. These mechanisms include access values, which are initialized to null, and default expressions. These simple mechanisms can be used to implement more complex structures.
- It is less safe. Depending on an explicit initialization operation adds a new opportunity for errors by forgetting to call the initialization operation. Also, these calls to the initialization operation would be separate from component declarations, increasing the likelihood of omission.
- It can make components harder to use. Instances of components would require an initialization call before use.
- It may have performance penalties. Performance is hindered if the initialization operation does nothing and the compiler does not optimize away the call.
- In the future initialization will be better supported by compilers. The Ada 9X draft mapping includes the ability to define an initialization routine which will automatically be called when variables are elaborated (as documented by Ada9X [1992, pp. 3-8]).

It can be easily argued that many of these points are also true for finalize, but the first point notes a key difference: the Ada default initialization mechanism can often be used to substitute for a separate initialization routine, but there is no equivalent Ada mechanism for finalization.

Section 8.3.1 of *Ada Quality and Style* specifically discourages initialization operations on every component unless necessary, especially because of the safety concerns. This remains unchanged in this candidate set of guidelines.

4.4 COMMON SOURCE

One guideline suggested by Edwards regards common source. Edwards recommends that common source should be located in a single location. This means that the Ada package specifications and bodies for multiple implementations of a single concept should come from a common source. Edwards notes that

possible mechanisms for doing so include the use of a preprocessor.

This guideline was removed from the originally proposed candidate set because there was controversy in the value of this guideline among the reviewers. Many reviewers believed that the condition where this guideline was useful was a specialized case. This condition is that more than one implementation of a given component would be used in the same application program. In the cases where this condition did occur, an alternative to this guideline from Edwards is to consider the components as two different (though related) components and not attempt to create a common source for the different components. These separate components would be relatively simple to manage, and comments could be included to note that other implementations. In addition, some reviewers believed that attempting to follow this common source guideline could make maintenance more difficult, since the common source might not stay common during maintenance.

The following text provides as additional information the rationale for considering this guideline.

Putting common source in a single location was believed to reduce maintenance costs, since changes need only occur in one place. Failing to do this could result in multiple maintenance,[3] the problem of having to locate and change the same code in more than one place. During the maintenance phase these copies might not stay equivalent when they should, creating the potential to cause errors that are difficult to trace.

There is a special case that occurs with developing reusable components, however: multiple implementations of a single component may be used in a program. A single component may have a number of different implementations. These implementations might differ in a number of ways, such as being bounded or unbounded in memory use, their average performance, worst-case performance, memory requirements, and accuracy.

If only one implementation will be used in a program, the implementation can be chosen using the compilation environment (for example, by choosing to compile only one of the implementations). In general, however, multiple implementations of a single component may be used in a program. Developers of reusable components must consider this possibility and, if relevant, make it possible to use more than one implementation of a component in the same program.

Putting common source in a single location can be more difficult when more than one implementation of a single component will be used in a program. In particular, Ada currently allows only one body to exist for each package specification. Even if multiple implementations are provided, only one can be linked into

3. Multiple maintenance is also called double maintenance, but this alternate term is misleading since the number of overlaps can be much greater than two.

an executable. This is true even if the Ada specification is a generic, in which case a single implementation must be chosen for *all* instantiations of that generic in a single program.

A user could recompile a different body to change the implementation, but this would restrict the user to a single implementation throughout the entire program.

One general purpose solution to this restriction is evident: to achieve multiple implementations in Ada, create a separate package specification/body pair for each implementation (each receiving a different package name). Changing implementations can be accomplished simply by altering the package name in the **with** clause (for examples, see Booch [1987, p. 55]).

A question then arises: how can common code (such as the specification and perhaps parts of the bodies) still come from a common source? One of the simplest solutions is the use of a preprocessor to generate the specification/body pairs from a single source. For program bodies the common code segments could be separated into generics which are used across several implementations (see Musser [1989] for an example).

A reasonable alternative might be to document in each component a cross-reference to the other related components. For more explanation and rationale for this guideline, see Edwards [1990, pp. 42-45], guideline 12.

4.5 PROCEDURE VARIABLE ENCAPSULATION

One guideline derived from Edwards recommends that if procedure (or function) variables are needed, a concept should be used which encapsulates their implementation. This derived guideline was later removed from the candidate set.

Recommending encapsulation is reasonable, but *Ada Quality and Style* already recommends information hiding as a general principle (section 4.1.4), and thus this derived guideline is already covered by a more general principle. Also, many reviewers of these candidate guidelines were concerned that specifically identifying encapsulation of procedure variables might imply that using procedure variables would automatically improve reusability. Many felt that the connection to reuse was somewhat obscure, as procedure variables are simply one possible mechanism for implementing approaches for improving reuse. It was decided that these approaches to reuse, such as table-drive programming (described in *Ada Quality and Style*, section 8.4.5), should be included in reuse guidelines instead of detailed discussions of specific mechanisms such as procedure variables.

Appendix B shows an example of a procedure variable concept that hides how procedure variables are implemented. This is followed by two possible implementations, one using calls through the C programming language and one using tasks. These are based on Edwards' work, but the code in the appendix does not depend on swapping as Edwards' does.

The implementation using the C programming language is not portable. In particular, it is implementation dependent (on the Ada compiler, the compiler version, and the platform) and context-dependent (it may work for some instantiations of ARG_TYPE but not others). For example, the Ada expression *p'address* need not be identical to the C expression **p*, the calling convention between C and Ada compilers need not be identical, the alignment may be different between the different compilers, and the pragma for calling to C might not be supported by a particular Ada compiler. Note that the procedure call from C to Ada must have matching compiler conventions. The advantage of the C implementation is that it will normally be faster than the tasking-based implementation.

For procedures with an arity (number of parameters) greater than one, there are two basic approaches. One is to pack and unpack data into the single parameter. An alternative approach, but which can have implementation and maintenance costs, is to modify the specification and implementation to handle more than one passed parameter. The implementation shown has a limited private ARG_TYPE; an alternative would to pass a non-private type (a record with the parameters).

5. CANDIDATE GUIDELINES

Table 3 shows the candidate guidelines for developing reusable Ada components. This table has two columns. The "short name" column contains the title of the guideline which is useful for referring to the guideline by name. The other column is the text of the guideline.

TABLE 3. Summary of Candidate Guidelines

Short Name	Guideline Text
Models	Use a model to describe reusable components.
Minimum profile	The minimum profile for a "concept" should be Copy, Is_Equal, Finalize, and Swap.
Finalize	Consistently use the Finalize operations of reusable components.
Self-composing component	As a test of the reusability of a component, consider composing the component with itself.
Arbitrary iterations	Arbitrary iterations should be constructible from primary operations.

Each guideline is described below. The guideline format is compatible with the structure and format found in *Ada Quality and Style*. This includes following the structure of headings (for example, **guideline** and **rationale**) and text formatting (for example, a bullet symbol precedes each guideline). This format is followed to make it easier for the reader to use these guidelines in conjunction with *Ada Quality and Style* as well as to facilitate the possible incorporation of these guidelines into later versions of the SPC and GE documents.

Each section begins with the short name and text of the guideline. This is followed by an **example**, **rationale**, and **notes**. There may also be a **comments** field which discusses how this guideline relates to the existing SPC or GE documents; this field would be omitted if the guideline were inserted into the GE or SPC guidelines. There is no **comments** field in the SPC or GE documents. Empty sections are omitted (as they are in *Ada Quality and Style*).

5.1 MODELS

guideline

- Use a model to describe reusable components.

rationale

Models are often used when working with complex objects. Using a model provides the following:

— a terminology as an aid to communication (by establishing a common point of reference).

— a framework for asking questions and for pointing out questions that should be asked.

— a way to reduce complexity by separating out a small number of important things to deal with at a time [Rumbaugh 1991].

There are a number of models, but no single model appears to be the best for discussing all reuse issues.

In the Department of Defense (DoD) there is a precedent for recommending a model without specifying which one. DoD Instruction 5000.2 requires the use of a process maturity model but does not specify a particular one.

example

This section describes a sample model termed the 3C model. The 3C model is relatively simple and its mapping to Ada is mostly straightforward. However, this is not intended to restrict practitioners to this one particular model.

The 3C model was developed at the Reuse In Practice Workshop in 1989 and was documented by Tracz [1989]. The name of the 3C model comes from the names of the three ideas on which it is based:

- The *concept*—what abstraction the component embodies. This includes not only the syntax for using the abstraction, but also its semantics. For example, a trivial *concept* might be a "stack" with the syntax and semantics of all its operations (such as push and pop).
- The *content*—how that abstraction is implemented. There may be more than one implementation of a *concept*, so there may be more than one *content* for a *concept*.
- The *context*—the software environment necessary to "complete" the component (including parameters provided by the component user).

To create a reusable component using the 3C model, separate *what* the component will do (the concept), *how* it will do it (the content), and *what external information* is necessary to tailor the component for use (the context).

There are two kinds of context in the 3C model—that of the concept (the *conceptual context*) and that of the content (the *implementation context*). A particular implementation may need additional context that is not relevant to other implementations of the same concept.

One simple mapping of the 3C model to Ada would implement the concept as a generic package specification, different contents as different package bodies for that specification, and the context as formal generic parameters of the package specification. This is not the only possible mapping; the key is to separate the three Cs and then use the appropriate Ada mechanisms to implement the component, given its environmental and performance constraints.

The benefits of describing reusable components using the 3C model in an Ada development environment are as follows:

- The 3C model separates the abstract ideas of concept, content, and context from how they can be implemented using Ada constructs. This frees developers from being constrained in their thinking to specific Ada constructs (such as generics).
- The 3C model stresses that the developer must consider the environment (context), make it visible, and separate it from both the concept and the content.
- The 3C model notes that there are contexts for both concept and content, and that implementations may require additional parameters beyond those needed, by the general concept.

The following are three possible misconceptions about reuse that a software developer using Ada can avoid by using the 3C model.

First, a developer may believe that Ada specifications by themselves define a concept. Ada specifications only formally define the syntax, and to implement a 3C concept the semantics must also be defined. Thus, additional information must be included with an Ada specification. These semantics might be specified informally using English. These semantics might also be specified formally using an annotation language such as Anna (as described by Luckham [1985]) or a formal specification language such as VDM (described by Jones [1986]) or Z (described by Spivey [1988 and 1989]).

Second, developers using Ada might not consider that multiple implementations (contents) of a concept are possible. Ada does not permit multiple bodies for a single specification within a single program, so component developers must work around this restriction when they wish to supply multiple implementations of a concept which can be used simultaneously in a single program.

Third, developers might not consider the possibility of a tailorable context for an implementation. Not only may concepts have a tailorable context, but

contents (implementations) may as well. A consideration is whether or not a particular implementation (content) has some additional parameters that should be accessible to the user.

For additional explanation and rationale of the 3C model and how it applies (in general) to Ada, see Edwards [1990, pp. 3-16]. Additional information on the 3C Model is given by Latour [1991a and 1991b] and Tracz [1989].

comments

Ada Quality and Style does not include any model or suggest that models should be used.

5.2 MINIMUM PROFILE

guideline

- The minimum profile for a "concept" should be Copy, Is_Equal, Finalize, and Swap.

example

```
type Item is [limited] private;
procedure Copy(from : in Item; into : in out Item);
function Is_Equal(left, right : in Item) return boolean;
procedure Finalize(i : in out Item);
procedure Swap(left, right : in out Item);
-- other operations.
```

rationale

A minimum profile helps make components more reusable because other components can then assume that at least these operations are available. Otherwise, if those operations are needed an alternative must be found.

Copy and Is_Equal are not automatically provided in Ada for limited private types, but many other components may depend on these operations. Thus these operations should be provided where possible. Note that the Booch components include these operations with the same semantics and operation names (as shown in Booch [1987]).

A Finalize operation checks if any resources should be released, and if they should, releases those resources. A Finalize operation could, for example, deallocate memory, close files or release semaphores. A Finalize operation should be included, even if it does nothing in the current implementation, so that later implementations can perform this operation and be assured that it will always be called when appropriate.

The reviewers of Ada 9X have recognized that finalization is an important operation. The capability to define a finalization operation (which is automatically called when a variable leaves its scope) is in the draft mapping of Ada 9X [1992, pp. 3-8]. For additional information, see the "finalize" guideline which specifies that this operation should be used.

Including the Swap operation in a minimal profile may seem unusual, but there are two reasons for including it in the minimum profile. First, a number of algorithms (such as sorting) depend on swapping and are far less efficient if they must simulate swapping using copying. Second, including a Swap operation makes it possible for other components to be built in the future using a different approach for data movement. There are good arguments that for reusable

components swapping is a better basic operator for moving data than copying, as described by Harms [1991] and Edwards [1990]. Including a swap operation makes it easy to use these components with either approach to moving data (i.e., copying or swapping).

It should be noted that the names of these operations are not as critical as the existence of the operations themselves. Ada's rename facility can be used to provide other names. Also, when used as parameters in generic instantiations, the operation names must be listed in any case.

This guideline should not be interpreted as requiring the use of limited private types. For more information on this limited private types, see *Ada Quality and Style* section 8.3.6 (private and limited private types). Ada limited private types provide complete control over a type, but can increase development time and decrease run-time performance. For example, Ada's predefined assignment, equality tests, and array indexing operations (with their convenient syntax and often efficient implementations) are not available for limited private types. A conscientious designer must carefully trade off the benefits of increased generality versus the cost of actually reusing the component, since in some cases a more general component may be harder to use.

For additional explanation and rationale of this guideline, see Edwards [1990, pp. 35-42] on guideline 10.

comments

Section 8.3.1 of *Ada Quality and Style* requires complete functionality and mentions Finalize, but neither mentions Swap nor gives such clear direction on a minimum profile.

5.3 FINALIZE

guideline

- Consistently use the Finalize operations of reusable components.

rationale

Finalization is an operation that returns dynamically allocated resources associated with a component. A Finalization operation may, for example, return dynamically allocated memory, close files, close (user interface) windows, or commit transactions that are associated with a component upon its deallocation. Data structures that require only static storage and do not allocate additional resources do not require finalization, and stack-based structures which do not allocate additional resources are finalized automatically. Among the collection of components used in a program, therefore, some may require explicit finalization while others may not.

Without finalization, a reused component that allocates dynamic storage, opens files or windows, etc., could cause a program to run out of resources, potentially halting its execution. The probability of problems occurring due to unreclaimed resources may be difficult to predict, as they depend on factors such as the usage pattern of the application and the resources available. Assuming that resources will be returned automatically can make reusable components considerably less reliable.

Finalization has a rather severe "ripple effect". Any data structure that incorporates a component that requires finalization will inherit the need for finalization, even though the new data structure would not otherwise require it. The new finalization operation may do nothing more than invoke its components' finalization operations; however, it is essential to propagate this service.

The irregularity between components that do and do not require explicit finalization and the ripple effect create a dilemma over when and where to provide finalization. The solution recommended by these guidelines is to ensure that all components have Finalize operations which are always invoked when the component leaves its scope. This improves the consistency of components and increases the possibility of composing larger components from smaller ones. The Finalize operation gives the component developer the opportunity to return resources without requiring the component user to know what those resources are.

When using Ada (as defined by ANSI [1983]), unfortunately, the user must always remember to invoke the Finalize operation explicitly when the component leaves its scope. Planned Ada 9X [1992, pp. 3-8] language revisions include a mechanism by which user-defined finalization operations will be invoked automatically, thus removing this responsibility.

Another possible solution to the dilemma is to provide two versions of a component, one with and one without finalization, and require the user to ensure that the version with finalization is used where appropriate. Components that do not invoke the finalization operations of incorporated components should be clearly documented as not providing this service.

Finalization complicates the development and use of components, thus increasing the cost of reuse. For some compilers, finalization may introduce extra run-time overhead in the case where (1) no explicit finalization is needed and (2) the compiler does not optimize away procedure calls that simply return. Thus, there are several trade-offs to be considered.

The Finalize operation is further described under the "Minimum Profile" guideline. The minimum profile guideline requires a component developer to *provide* a Finalize operation, while this guideline requires that Finalize always be *used*.

For additional explanation and rationale of this guideline, see Edwards [1990, pp. 35-42] on guideline 7.

comments

This guideline might be implied from *Ada Quality and Style*, section 8.3.1 (complete functionality), but is not clearly stated.

5.4 SELF-COMPOSING COMPONENT

guideline

- As a test of the reusability of a component, consider composing the component with itself.

rationale

For a component to work in various contexts, its parameters and exported operations must be robust. Thus, it is important to have ways of testing this robustness.

One test of robustness is to consider composing a component with itself. Many components depend on other components as parameters (for example). Self-composing a component means that the component is used as its own parameter. Attempting to do this simultaneously tests how well the component can (1) use other components and (2) be used by other components. For a number of components this capability is itself important, for this makes it possible to use these components as "building blocks" to be combined in different ways to implement more complicated components.

As a simple example, consider a "stack" component. You should be able to simply create a "stack of stacks". If you implement this stack as a generic abstract data type (ADT),[4] you should be able to take the exported type and operations from one instantiation and use them to instantiate a second stack package.

It is important to note that a component might not ever be *used* when composed with itself. This guideline is simply a *test* of reusability to help the developer check for missing parameters or operations. This test is likely to be appropriate for small container components which are intended to be combined with similar kinds of components. For some components this test may be inappropriate.

For additional information about this guideline, see Edwards [1990, pp. 35-42] on guideline 9.

notes

Implementing a self-composed component may reveal problems that might otherwise be missed. However, even considering the issue may help identify problems.

---

4. As defined in *Ada Quality and Style* section 8.3.4, "Using Generic Units for Abstract Data Types".

5.5 ARBITRARY ITERATIONS

guideline

- Arbitrary iterations should be constructible from primary operations.

rationale

A component can be used in more applications if arbitrary iterations can be constructed from the operations provided. If this cannot be done, new iterations may require modification of the component itself. Each modification would then require time to design, code, and test, whereas a more general iteration capability would eliminate this need entirely. This guideline only applies to components where iteration is appropriate.

For additional information on this guideline, see Edwards [1990, pp. 70-82] on guideline 18.

comments

Ada Quality and Style's sections 8.3.5 (which describes iterators) and 8.3.1 (which requires complete functionality) might imply this when combined, but neither explicitly state this. Section 8.3.5 requires active iterators but does not state that arbitrary iterations should be constructible from the existing operations.

APPENDIX A

GUIDELINE ALLOCATION

Every guideline from Edwards' document has been considered. The following table shows where each guideline has been allocated in this document. The first column shows its guideline number in Edwards' document. The second column includes a short descriptive name of the guideline; no such names were given in Edwards, so these names have been created for the purpose of this document. Each of the guidelines is allocated to one of the following categories: SPC (it overlaps with a guideline in *Ada Quality and Style*), Edwards (it is subsumed by another guideline in Edwards), Study (it requires further study), or Candidate (it has been included in the candidate list of guidelines).

This list shows 25 guidelines. Edwards only number 21 guidelines, but two additional guidelines were implied (with guideline numbers "N/A") and two others (guidelines 7 and 10) have been split to allocate them to different categories, bringing the total to 25.

TABLE 4. Allocation of Edwards' Guidelines

Guide-line No.	Short Name	SPC	Edwards	Study	Candidate
1	Single Specification	●			
2	Only One Abstraction	●			
3	Horizontal Coupling	●			
4	Complete Operation Set	●			
5	Limited Private	●			
6	Initialize and Finalize		●		
7a	Initialize			●	
7b	Finalize				●
8	Context types	●			
9	Self-Composing Component				●
10a	Minimum Profile: Initialize			●	
10b	Minimum Profile				●
11	Separate Package Implementations	●			
12	Common Source			●	
13	Structural Sharing			●	
14	Using Swap			●	
15	Define Swap		●		
16	Constructors and Selectors with Swap			●	
17	Copy and Is_Equal		●		
18	Arbitrary Iterations				●
19	Iteration Types	●			
20	Random Access Iterator	●			
21	Save/Restore			●	
N/A	Models				●
N/A	Procedure Variable Encapsulation			●	

APPENDIX B

PROCEDURE VARIABLE EXAMPLE

This appendix presents a procedure variable concept that hides the implementation of procedure variables. It is intended as an example of a procedure variable component. This is followed by two possible implementations, one using tasks (for portability) and one using calls through C.

1. PROCEDURE VARIABLE SPECIFICATION

```
-- Procedure variable concept

-- This defines a procedure variable type (PROCEDURE_VARIABLE).
-- Variables of type PROCEDURE_VARIABLE may be set to the value
-- of a procedure, and the variable may later be used to call whatever
-- procedure the variable was last set to.
-- When called the procedure will be passed a variable of type ARG_TYPE.

-- This component is based on the one in the paper
-- ''An Approach For Constructing Reusable Software Components in Ada''
-- by Stephen Edwards.  IDA Paper P-2378, Sept. 1990.
-- Alexandria, VA: Institute for Defense Analyses (IDA).
-- Unlike the component in Edwards' paper this component does not depend
-- on swapping.

generic
 -- ARG_TYPE is the type which will be passed to the procedure variable.
 type ARG_TYPE is limited private;
 with procedure Initialize(Data : in out ARG_TYPE);
 with procedure Finalize  (Data : in out ARG_TYPE);

package Procedure_Variable_Abstraction is
 type PROCEDURE_VARIABLE is limited private;

 procedure Initialize(Data : in out PROCEDURE_VARIABLE);
   -- This initializes a procedure variable to the conceptual
   -- value "NULL."
   -- This must be executed for each PROCEDURE_VARIABLE declared.
 procedure Finalize  (Data : in out PROCEDURE_VARIABLE);
   -- This releases all resources associated with PROCEDURE_VARIABLE.
   -- It must be executed on each PROCEDURE_VARIABLE before that variable
```

```
  -- goes out of its defining scope.
 procedure Swap(left  : in out PROCEDURE_VARIABLE;
                right : in out PROCEDURE_VARIABLE);
 function Procedure_Variable_Is_Null(PV : in PROCEDURE_VARIABLE)
     return BOOLEAN;
 procedure Reset_Procedure_Variable(PV : in out PROCEDURE_VARIABLE);

 generic
    with procedure P(a: in out ARG_TYPE);
 package Procedure_Definer is
    procedure Set_Procedure_Variable_To_P(PV: in out PROCEDURE_VARIABLE);
 end Procedure_Definer;

 procedure Invoke_Procedure(PV : in PROCEDURE_VARIABLE;
                            Arguments : in out ARG_TYPE);

 UNINITIALIZED_PV : exception;

private
 type PV_BLOCK;
 type PROCEDURE_VARIABLE is access PV_BLOCK;
end Procedure_Variable_Abstraction;
```

2. PROCEDURE VARIABLE IMPLEMENTATION USING TASKING

```
-------------------------------------------------------------------------------
-- @(#)proc_var.b1.a    1.2       9/6/90
-------------------------------------------------------------------------------
with unchecked_deallocation;
package body Procedure_Variable_Abstraction is

    task type go_between_type is
      entry in_args    (a  : in out arg_type);
      entry out_args   (a2 : in out arg_type);
      entry return_args(a3 : in out arg_type);
    end go_between_type;

    type procedure_type is access go_between_type;
    null_procedure : procedure_type := null;
    type pv_block is record
      pv : procedure_type := null_procedure;
    end record;

    procedure initialize(data : in out procedure_variable) is
    begin
      if data /= null then
          finalize(data);
      end if;
      data := new pv_block'(PV => null_procedure);
    end initialize;

    procedure finalize(data : in out procedure_variable) is
      procedure free is new unchecked_deallocation(
          pv_block, procedure_variable);
    begin
      if data /= null then
          free(data);  -- assigns data = null after deallocating space
      else
          raise UNINITIALIZED_PV;
      end if;
    end finalize;

    procedure swap(left  : in out procedure_variable;
                   right : in out procedure_variable) is
      temp : procedure_variable := left;
    begin
      -- The normal "swap" implementation.  Note that it runs
      -- in "constant" time, regardless of the size of a
```

```
  -- PV_BLOCK, so the representation of a procedure variable
  -- can be altered without affecting its efficiency.
  left := right;
  right := temp;
end swap;

function procedure_variable_is_null(pv : in procedure_variable)
  return boolean is
begin
  if pv = null then
      raise UNINITIALIZED_PV;
  else
      return pv.pv = null_procedure;
  end if;
end procedure_variable_is_null;

procedure reset_procedure_variable(pv : in out procedure_variable) is
begin
  if pv = null then
      raise UNINITIALIZED_PV;
  else
      pv.pv := null_procedure;
  end if;
end reset_procedure_variable;

package body Procedure_Definer is

  task shell is
      entry receive_go_between(gb_holder : in procedure_type);
  end shell;

  go_between : procedure_type := new go_between_type;

  procedure set_procedure_variable_to_P(pv : in out procedure_variable) is
  begin
      if pv = null then
        raise UNINITIALIZED_PV;
      end if;
      pv.pv := go_between;
  end set_procedure_variable_to_P;

  task body shell is
      gb : procedure_type;
      a  : arg_type;
  begin
      initialize(a);  -- set it to a valid initial value
      accept receive_go_between(gb_holder : in procedure_type) do
        gb:= gb_holder;
      end receive_go_between;
      loop
```

```
            -- swap the requested argument value into A
            gb.out_args(a);
            -- invoke the actual procedure
            p(a);
            -- swap the (possibly altered) value back to the caller
            gb.return_args(a);
          end loop;
          -- This point is unreachable, but for completeness,
          -- clean up when done :
          finalize(a);
        end shell;

    begin
      shell.receive_go_between(go_between);
    end Procedure_Definer;

    procedure invoke_procedure(pv : in     procedure_variable;
                               a  : in out arg_type) is
    begin
      if pv = null then
          raise UNINITIALIZED_PV;
      elsif pv.pv /= null_procedure then
          pv.pv.in_args(a);
      end if;
    end invoke_procedure;

    task body go_between_type is
    begin
      loop
          accept in_args(a : in out arg_type) do
            -- accept input to procedure P
            accept out_args(a2 : in out arg_type) do
                -- put P's arg into a2 so SHELL task can see it
                swap(a2, a);
            end out_args;
            accept return_args(a3 : in out arg_type) do
                -- take output from SHELL task and put it back
                -- in A to be passed back to INVOKE_PROCEDURE.
                swap(a, a3);
            end return_args;
          end in_args;
      end loop;
    end go_between_type;

end Procedure_Variable_Abstraction;
```

3. PROCEDURE VARIABLE IMPLEMENTATION USING INTER-LANGUAGE CALL

```
--------------------------------------------------------------------------------
-- proc_var.b2.a
--------------------------------------------------------------------------------
with System, Unchecked_Conversion, Unchecked_Deallocation;
package body Procedure_Variable_Abstraction is

    type arg_type_ptr is access arg_type;
    subtype procedure_type is system.address;
    null_proc : integer := 0;
    null_procedure : constant procedure_type := null_proc'address;
    type pv_block is record
      pv : procedure_type := null_procedure;
    end record;

    procedure initialize(data : in out procedure_variable) is
    begin
      if data /= null then
          finalize(data);
      end if;
      data := new pv_block'(PV => null_procedure);
    end initialize;

    procedure finalize(data : in out procedure_variable) is
      procedure free is new unchecked_deallocation(
          pv_block, procedure_variable);
    begin
      if data /= null then
          free(data);  -- assigns data = null after deallocating space
      else
          raise UNINITIALIZED_PV;
      end if;
    end finalize;

    procedure swap(left  : in out procedure_variable;
                   right : in out procedure_variable) is
      temp : procedure_variable := left;
    begin
      -- The normal "swap" implementation.  Note that it runs
      -- in "constant" time, regardless of the size of a
      -- PV_BLOCK, so the representation of a procedure variable
      -- can be altered without affecting its efficiency.
      left := right;
```

```
  right := temp;
end swap;

function procedure_variable_is_null(pv : in procedure_variable)
  return boolean is
  use system;
begin
  if pv = null then
      raise UNINITIALIZED_PV;
  else
      return pv.pv = null_procedure;
  end if;
end procedure_variable_is_null;

procedure reset_procedure_variable(pv : in out procedure_variable) is
begin
  if pv = null then
      raise UNINITIALIZED_PV;
  else
      pv.pv := null_procedure;
  end if;
end reset_procedure_variable;

package body Procedure_Definer is

  procedure p_wrapper(aa : in system.address) is
      function from_sa is new unchecked_conversion(
        system.address, arg_type_ptr);
      a : arg_type_ptr := from_sa(aa);
  begin
      p(a.all);
  end p_wrapper;

  procedure set_procedure_variable_to_P(pv : in out procedure_variable) is
  begin
      if pv = null then
        raise UNINITIALIZED_PV;
      end if;
      pv.pv := p_wrapper'address;
  end set_procedure_variable_to_P;

end Procedure_Definer;

procedure invoke_procedure(pv : in     procedure_variable;
                           arguments  : in out arg_type) is
  procedure c_invoke_hook(a : in system.address;
                          p : in system.address);
  pragma interface(c, c_invoke_hook);
  use system;
begin
```

```
      if pv = null then
          raise UNINITIALIZED_PV;
      elsif pv.pv /= null_procedure then
          c_invoke_hook(arguments'address, pv.pv);
      end if;
    end invoke_procedure;

end Procedure_Variable_Abstraction;


/* C function to invoke procedure ''procedure_variable'' with ''arguments'' */
void c_invoke_hook(arguments, procedure_variable)
 int *arguments;
 void (*procedure_variable)();
{
 (*procedure_variable)(arguments);
}
```

REFERENCES

[Ada9X 1992] Office of the Under Secretary of Defense for Acquisition. *Ada 9X Mapping Document Volume 1: Mapping Rationale*. March 1992. Cambridge, MA: Intermetrics, Inc.

[AdaIC 1991a] *Ada Information Clearinghouse (AdaIC) Newsletter*. September 1991. Vol IX, No. 3. Lanham, MD: IIT Research Institute.

[AdaIC 1991b] *Ada Information Clearinghouse (AdaIC) Newsletter*. December 1991. Vol IX, No. 4. Lanham, MD: IIT Research Institute.

[Ahlberg 1992] Ahlberg, Axel. 1992. Personal Communication.

[ANSI 1983] ANSI/MIL-STD-1815A-1983. 1983. *Reference Manual for the Ada Programming Language*.

[Baldo 1990] Baldo, James. April 1990. *Reuse In Practice Workshop Summary*. IDA Document D-754. Alexandria, VA: Institute for Defense Analyses.

[Booch 1987] Booch, Grady. 1987. *Software Components with Ada*, Second Edition. Menlo Park, CA: Benjamin/Cummings.

[Edwards 1990] Edwards, Stephen. 1990. *An Approach for Constructing Reusable Components in Ada*. IDA Paper P-2378. Alexandria, VA: Institute for Defense Analyses.

[Edwards 1992] Edwards, Stephen. 1992. Personal Communication.

[GE 1991] General Electric Aerospace. August 19, 1991. *GPALS Software Standards*. CDRL A095. Blue Bell, PA: General Electric Aerospace.

[Harms 1991] Harms, Douglas E. and Bruce W. Weide. 1991. "Copying and Swapping: Influences on the Design of Reusable Software Components." *IEEE Transactions on Software Engineering*. May 1991. Vol 17, No. 5, pp. 424-435.

[Jones 1986] Jones, C.B. 1986. *Systematic Software Development Using VDM*. Prentice-Hall International.

[Latour 1991a] Latour, Larry, Tom Wheeler, and Bill Frakes. 1991. "Descriptive and Predictive Aspects of the 3Cs Model: SETA1 Working Group Summary." *Ada Letters*, Spring 1991, pp. 9-17.

[Latour 1991b] Latour, Larry. 1991. "A Methodology for the Design of Reuse Engineered Ada Components". *Ada Letters*, Spring 1991, pp. 103-113.

[Luckham 1985] Luckham, D. and F.W. von Henke. March 1985. "An Overview of Anna, A Specification Language for Ada". *IEEE Software*: 9-22.

[Musser 1989] Musser, David R. and Alexander A. Stepanov. 1989. *The Ada Generic Library: Linear List Processing Packages.* New York, NY: Springer-Verlag.

[Rumbaugh 1991] Rumbaugh, James et al. 1991. *Object-Oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall.

[SDIO 1992] Strategic Defense Initiative Organization. March 1992. *SDIO Software Policy*. SDIO Directive No. 3405 (Revision 1).

[SPC 1989] Software Productivity Consortium (SPC). 1989. *Ada Quality and Style: Guidelines for Professional Programmers.* ISBN 0-442-23805-3. New York, NY: Van Nostrand Reinhold.

[SPC 1991] Software Productivity Consortium (SPC). 1991. *Ada Quality and Style: Guidelines for Professional Programmers.* Version 02.00.02, SPC-91061-N. Alexandria, VA: Defense Technical Information Center (DTIC).

[Spivey 1988] Spivey, J. M. 1988. *Understanding Z—A Specification Language and Its Formal Semantics.* Cambridge Tracts in Computer Science 3. Cambridge University Press, 1988.

[Spivey 1989] Spivey, J. M. 1989. *The Z Notation: A Referen · Manual.* Prentice-Hall International.

[Tracz 1988] Tracz, Will J. 1988. *Tutorial: Software Reuse: Emerging Technology*. Washington, DC: IEEE Computer Society Press. IEEE Catalog Number EH0278-2. ISBN 0-8186-0846-3.

[Tracz 1989] Tracz, W.J. 1989. "Implementation Working Group Report." *Reuse In Practice Workshop Summary*. James Baldo. IDA Document D-754. Alexandria, VA: Institute

for Defense Analyses.

[Weide 1986] Weide, Bruce. January 1986. *A Catalog of OWL Conceptual Modules*. Columbus, Ohio: The Ohio State University.

ACRONYMS

3C	Concept, Content, and Context
ACM	Association for Computing Machinery
ADT	Abstract Data Type
AJPO	Ada Joint Program Office
DoD	Department of Defense
GE	General Electric
GPALS	Global Protection Against Limited Strikes
IDA	Institute for Defense Analyses
SDIO	Strategic Defense Initiative Organization
SEI	Software Engineering Institute
SPC	Software Productivity Consortium